NASA/CR-1998-207636
ICASE Interim Report No. 31

Parallel PAB3D: Experiences with a Prototype in MPI

Fabio Guerinoni
Virginia State University

Khaled S. Abdol-Hamid
Analytical Services & Materials, Inc.

S. Paul Pao
NASA Langley Research Center

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA
Operated by Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract NAS1-19480

April 1998

PARALLEL PAB3D : EXPERIENCES WITH A PROTOTYPE IN MPI

FABIO GUERINONI *, KHALED S. ABDOL-HAMID †, AND S. PAUL PAO ‡

Abstract. PAB3D is a three-dimensional Navier Stokes solver that has gained acceptance in the research and industrial communities. It takes as computational domain, a set disjoint blocks covering the physical domain. This is the first report on the implementation of PAB3D using the Message Passing Interface (MPI), a standard for parallel processing. We discuss briefly the characteristics of the code and define a prototype for testing. The principal data structure used for communication is derived from preprocessing "patching". We describe a simple interface (COMMSYS) for MPI communication, and some general techniques likely to be enconuntered when working on problems of this nature. Last, we identify levels of improvement from the current version and outline future work.

Key words. Message Passing Interface (MPI), Navier-Stokes solver, structured meshes, broadcasting, point-to-point communication

Subject classification. Computer Science

1. Introduction . Parallel processing has been a trend in the aerospace industry for more than a decade. A number of systems have emerged which run in a number of processors. Significant examples are the ENS3D, recently ported to the Intel Paragon and Pratt & Whitney's NASTAR [6]. Other examples can readily be found in conference proceedings, for example [5].

Many of this codes were designed in the late 80's when there was a trend for large shared-memory systems, like the Cray Y-MP or the NEC SX-4. In consequence, many of these codes were designed to run as a number of more or less independent tasks, implicitly communicating by using shared memory. As the limitations in term of scalability and costs starts showing up in shared memory systems, the use of distributed memory in the form of massively parallel processing systems or clusters of workstation became an standard trend. The widespread and free availability of systems like PARMACS, PVM, and more recently, MPI and MPI-2 contributed to the process.

In the transition, the fact that the codes were written for shared memory multiprocessors simplified the task of switching to distributed memory system. The original systems takes care of parallel I/O and computation; the programmer task becomes implementing the *communication*.

Parallelizing an application from a sequential code is more complicated, since one has to take into account other issues besides communication. In the first of the outcomes of our project, we show that the

* Department of Mathematics, Virginia State University, P.O. Box 9068, Petersburg, VA 23806. This research was supported by the National Aeronautics and Space Administration under Contract No. NAS1-19480 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681-0001.

† Analytical Services & Materials, Inc. Hampton, VA 23666

‡ Configuration Aerodynamics Branch NASA Langley Research Center, MS 499, Hampton, VA 23681-0001

tranformations are feasible provided a limited amount of the resources. The key to success, in our opinion, was to have a well-defined *prototype* with the key characteristics:

- The problem must be of realistic size
- Standard options
- Limited, but essential functionality

This is the first report on the parallelization of a protype for PAB3D. Lessons learnt here will be used to provide a better version of PAB3D for parallel processing. We will restrict ourselves to describe some techniques used and suggesting directions for continuing improvements. Our prototype is necessarilly rough-hewn. Some elementary concepts in parallel processing such as "load balancing" or "speed-up" have been purposedly left out, as there is still significant amount of work to be done. The important achievement is that the prototype runs for a relative large number of heterogeneous processors, and produces correct residuals for a single, but realistic problem.

The report is organized as follows. Following a summary description of PAB3D, we go on to the describe the principal parallel implementations decision, chosen primarily because the simplicity of its development. A brief description of the MPI implementation follows.

Section 5 is the core of the report. On it we describe some adhoc techniques that has proven very useful in our particular case of PAB3D and certainly extends to other applications. As it turns out, the key issue here is the use of a data structure designed for the *sequential* block-cell connectivity and the use of an interface to MPI which we call COMMSYS (for COMMnication SYStem). We conclude with suggestions for future work.

2. PAB3D Characteristics.

2.1. Brief description of Code . The PAB3D code (currently in its version 13) is a three dimensional Reynolds-averaged Navier-Stokes (RANS) solver. It was initially developed in 1986 by Khaled S. Abdol-Hamid for supersonic jet exhaust flow analysis. After enhancements to the code since that time by the use of multi-block/multi-zone techniques, it has become a general purpose Navier-Stokes code for complex aerodynamic and propulsion integrated configurations [1]. This code has several schemes for the RANS including turbulence models, and multi-block capability. Here are some of them:

1. Treatment of Convection terms: upwinding is used, among which is possible to choose the following variants.

- Roe numerical flux
- van Leer flux splitting
- van Leer implicit

2. Limiters, required to to prevent oscillations in high order methods near shocks. A number of strategies are incorporated in the code

- Van Albada
- Sweby's min-mod
- S-V (Spekreijse-Venkat)
- Modified S-V

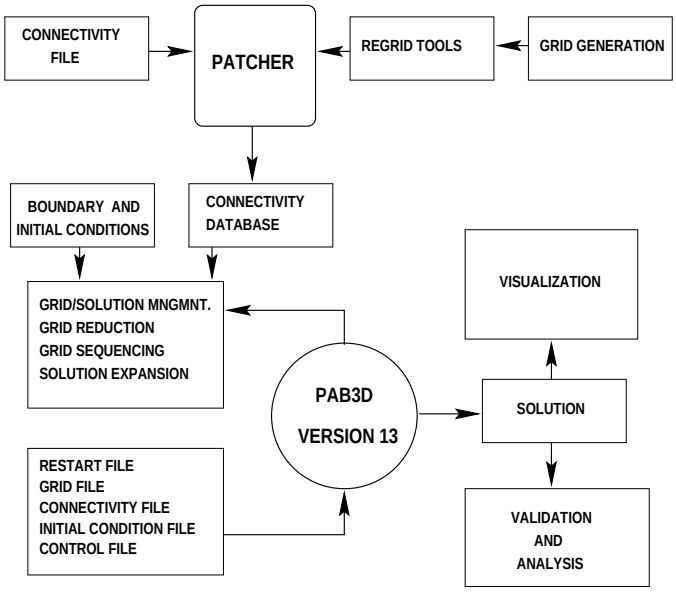

FIG. 2.1. *Use of PAB3D, Version 13, and related programs and files*

3. Treatement of Viscosity terms: is as usual centrally approximated. There is a wide choice for the cross terms

- j-thin layers, k-thin layers
- jk-uncoupled
- jk-coupled

4. Turbulence model: is completely independent of the solver part. A wide variety of models have been implemented into the code, including a number of algebraic Reynolds-stress models.

- Two-equation k-epsilon model
- Shih-Zhu-Lumley
- Gatski-Speziale
- Grimaji

Other important features include the ability to deal with real gas equations of state and several compressibility turbulence correction methods. The code accepts flows involving non-reacting multi-species from which "effective" viscosity and other pararameters are computed. It is also possible to specify other boundary conditions.

A schematic view of PAB3D and its relation with other programs and files is shown in Figure 2.1. A detailed description of this code with emphasis on the turbulence models can be found in [3, 2].

2.2. The Patching. The most significant improvement ot to the PAB3D code occurred in 1990 when "conservative patching" was introduced to the code [4]. This allowed the multi-block/multi-zone structure of the code by the creation of new cells at the interaces. A group of such cells is a *patch* ,

called a *piece* in this report. The amount of overlapping of the cells ratios are stored in arrays so that space-integrated fluxes are computed conservatively.

Later, the patches databases were expanded and improved. Version 13 of the code (current) contains three significant data bases:

- **IPCB(piece,pieceinfo)** : A *global patch* database, Depending on the value of pieceinfo, the corresponding information for the piece is provided. Four our purposes wet set pieceinfo so as to get:
  * the number of cells in the piece
  the face/block where the piece belongs
  adjacent face/block

- **IPTF(block,blockinfo)** : The *block database* which provides, dimensions for the structured block, and after the patching is done, the pathces involved in the exchange of information
  number of pieces
  list of local pieces
  list of adjacent local pieces

- **IPCBL(block,face,localpiece)** : The *piece local-to-global mapping*. A face in a block may contain several pieces, identified by a local number. This array provides the corresponding global piece number to access the database if required.

These are the only arrays in volved in the commmunication part of the parallel system, as explainge in Section 5.

2.3. The Prototype Problem . The computational grid for the parallel process test case has nine blocks and a total of 1.29 million grid points. The physical model is a convergent-divergent Mach 2 nozzle which was designed for the Jet Noise Laboratory at NASA Langley. The computational grid characteristics are given in the following table:

nb	idm	jdm	kdm	nbt	Description
1	61	33	53	106689	interior of nozzle
2	61	33	61	122793	exterior of nozzle
3	65	33	113	242385	downstream of nozzle
4	97	33	113	361713	downstream of Block 3
5	97	33	113	361713	downstream of Block 4
6	61	17	17	17629	cartesian core 1
7	65	17	17	18785	cartesian core 2
8	97	17	17	28033	cartesian core 3
9	97	17	17	28033	cartesian core 4

The computational grid describes one quarter of the physical nozzle and its ambient environment. The high pressure plenum chamber and nozzle flow acceleration path to Mach 2 at the nozzle exit is contained

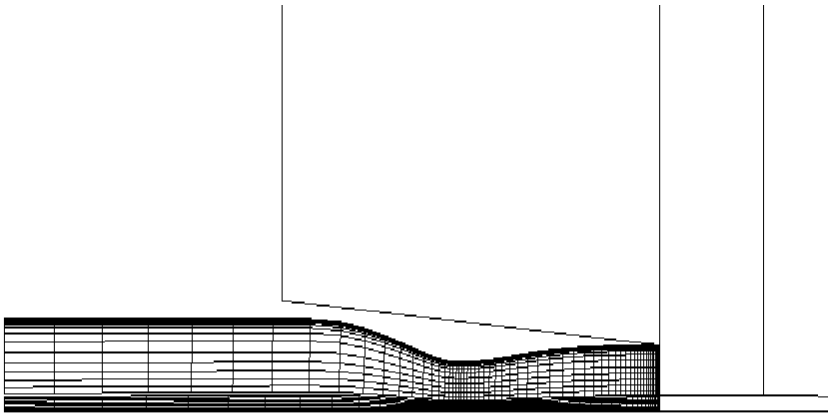

FIG. 2.2. *Blocks near nozzle. Block 1 is the interior of the nozzle*

in block 1. The ambient flow which provides a laboratory-type nozzle exit environment is described in blocks 2 through 5. The grid topology for these five block are cylindrical. In order to eliminate the polar singularity at the axis of the nozzle interior and the jet exhaust plume flow path, the flow domain surrounding the axis of symmetry is covered by cartesian grids of blocks 6 through 9. The connectivity between block faces are described by one or more patched interface data tables for each pair of block interfaces. The tables are generated automatically (patching) by a block interface connectivity preprocessor, known simply as the "Patcher" utility. This example is chosen both for the simplicity of the physical flow configuration, and the moderate complexity of a multiblock grid with general connectivity requirements. The block sizes come in two groups: five blocks with an average of 250,000 grid points and four blocks with an average of 25,000 grid points. These can be used to test workstation clusters with different speed and memory capacities. Some of these blocks are small enough such that multiple processes can be initiated on a single workstation under the MPI system.

As required by the prototype conditions listed above, the numerical techniques are fairly standard:

- Ideal gas simulation
- Standard k-e models
- Roe's flux differencing upwind scheme
- Third order interpolation
- Coupled viscous terms in the j-k plane

3. Parallel Implementation Decisions.

3.1. Model of Computation. From the beginning, it was clear the parallelism would be at the block (spatial) level. That is, the each process would be in charge of a block. For this type of computations,

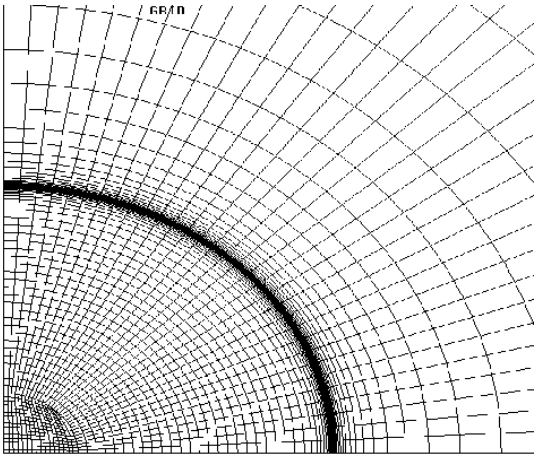

FIG. 2.3. *A cross section of the prototype problem*

one finds in the literature two types of well-establisehd approaches.

In the *master-slave* approach a processsor designated to coordinate the *spawning* of tasks on the other available process and possibly required to handle all the asynchronous communication between the slave processor in a point-to-point fashion. When a slave needs to send/receive data, it does through the master process, which receives, the message buffers it and resends appropriate. Thus the master synchronizes the operation of the slaves, but it does not participate in the computations.

One of the main differences between the original MPI (MPI-1) and one of its predecessor PVM, is the inability of the former to spawn a task. Tasks are started at user interface. In a master-slave approach this requires having two executables. Since some of the goals of the project was to mantain the simplicity of the sequential code, we decided that the master-slave approach was not appropriate. Thus, we incline in favor of the *graph* model, in which each processor does the same type of task as any other node. All nodes are in charge of communication and computation.

3.2. Data Distribution. In an optimal parallel implementation of a code, not only the computational work must be distributed evenly among processors, but data must be distributed as well. This is called a *shrunken block* model. On the other hand keeping in each process a full adressing space, is called the *full block* model.

However, size of code was not of premium concern. Most of the data is devoted to storing the unknown variables, and the global grid coordinates. As we were confindent that we would get at least as many processors as there are blocks, we opted for the full block model. Such approach, easier to implement, is often used for preliminary versions of parallel code, as is now the case. Figure 3.2 shows a schematic of the alternatives.

4. The MPI implementation: LAM. There are several implementations of MPI, all available from free. Among the most widely developed and more robust are the Argonne MPICH and the Ohio

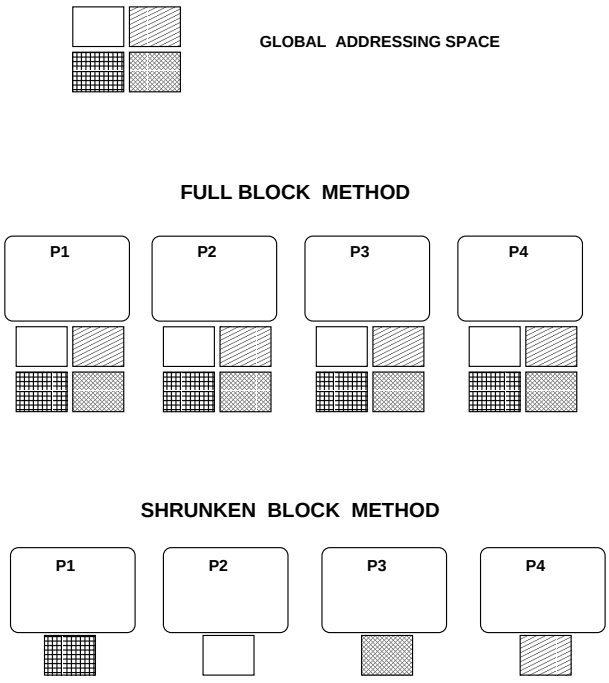

FIG. 3.1. *Data distribution among processors. Starting from a sequential code the full block is easier to implement*

Supercomputer Center Local Area Multicomputer (LAM) [7]. For the parallelization of PAB3D we have used the former in its current version 6.1.

The LAM version of MPI has some extensions. For example, the original MPI [9] does not allow for dynamic process spawning. LAM includes extensions to do this. These should not be confused with the added functionality of MP1-2, still in the making. In addition, LAM comes with utility commands that allow process/processor control through a file called *application schema* and configuration control a *process schema*.

Some commands allow probing of the status of the remote hosts, described by the process schema. For example, the command *mpitask* shows the following output

```
TASK (G/L)          FUNCTION      PEER|ROOT  TAG    COMM    COUNT   DATATYPE
0/0 pab3d           Bcast         0/0               WORLD* 6438865 REAL
8/8 pab3d           WaitAll       0/0               WORLD* 256 REAL
3 pab3d             <running>
1/1 pab3d           Bcast         0/0               WORLD* 6438865 REAL
2/2 pab3d           Bcast         0/0               WORLD* 6438865 REAL
```

The display information shows, the processor and name of process, its MPI function at the moment. The status *¡running¿* indicates non-MPI activity. The PEER and TAG fields involves point-to-point

communication.COMM is the communicator involved (an MPI notion to delimit the group of process with which exchange messages). COUNT indicates the size of message and DATATYPE its type.

MPI is complex enough, but most applications require only a dozen commands or so. Besides the control, initialization termination and identification, the most important commands are for *point-to-point* communication and *collective* operations, [8]. A short sampler of typical operations in each category follows.

Point-to-point communication

* MPI_RECV
* MPI_IRECV
* MPI_SEND

The MPI_SEND in four flavours.They are "blocking" in the sense that the call will not return until some "event" has happened. Similarly, MPI_RECV is blocking, but MPI_IRECV is not.

Collective communication

* MPI_BCAST
* MPI_GATHER
* MPI_REDUCE

MPI_BCAST is used to distribute information among all process in the communicator. MPI_GATHER collects information from other processes. MPI_REDUCE might be used in conjuction with arithmetic operations to obtain a result in a single process but which involves all process, as when doing a scalar product.

5. An approach to message passing computations. After the defining the data orgainization, the actual implementation and incorporation the MPI calls was carried out in four distinct phases, M1 through M4. Source code with the MPI calls corresponding to a phase was identified by providing an appropriate suffix. In each step we tried to include only the routines necessary for their complete and independent testing.

An important technique that proved very useful, was to develop a communication subsystem, which we call COMMSYS, which was tested independently of the main code. In the part of the commmunication step, as described below, the implementation reduced to write an interface to COMMSYS. The original code remain virtually unchanged.

5.1. Phase M1: Start-up. It is widely acknowledged that the single characteristic of the sequential codes that do most to prevent parallelizaition is the issue of I/O. In large engineering software projects, developed over a number of years, it is natural that modifications and improvements are done essentially in the computation part of the code. Developers tend to add I/O statements to the code generously, without any consideration for parallel processing. This is either done for genuine reasons or, in many cases, for simple debugging purposes.

The resulting code is extremely difficult or expensive to parallelize. The reason being is that each I/O

cannot be executed by more than a single processor. Furthermore, if code is modified so as to ensure execution by a single processor, input staments must be followed by expensive broadcasts. An exception to this is when the code written from the start with *distributed I/O*. This is an active area of research these days. Eventually, all parallel codes must be able to deal with distributed I/O for efficiency.

The principal goal of phase M1 was to ensure that all computing processors have the proper data at the beginning of the main computational routine. In the prototype PAB3D, this was identified as routine *solver*. Unfortunately, there is still some I/O within this routine, fact that complicated things further. Some of these are avoidable, some others are not. We will discuss more on this on the final step.

In short, the goals of M1 were two:

- At the starting of the computational part, each process must have *exactly* the same information in its arguments, local variables and commons as the single process do.
- Run with enough processors, so as to accomodate each main computational routine.

The first goal was achieved by broadcasting after input operation. Dummy arrays (whose size is determine at run-time) were dealt in full scale rather than limiting it to actual size. Thus, the constants of the arrays dimensions were used in the broadcast calls, rather than the actual size.

By doing limited (actual size) broadcasting, turnaround times would have been less and thus would have speed development. But as our project was completely full-scale, application-oriented, we could not afford to take risk by testing a reduced-scale prototype, only to find out later that it will not scale properly.

As it turn out, and in spite that we developed ad-hoc utilities for the purpose, this task was by far the most delicate and time consuming of the overall project. The straighforward technique of restricting the I/O statement to one processor and then broacast in the input case, was ruled out from the beginning. As the PAB3D system contains about 1600 I/O FORTRAN statements, this approach was clearly out of the question. A new technique was needed.

At higher level routines, where subroutines tend to be called within the same context and functionality, a very useful approach was to partition the I/O intensive or I/O subroutine intensive, in in two type of segments: G and B segments.

B-segments were constructed according to the following characteristics

1. There is no branching into the segment. The idea is that the segment will execute in a single process (to avoid I/O conflicts), and thus code branching into the segment risks being improperly executed.
2. The segment must maximize the number of input operations, while at the same time must contain a minimal of computation in between.
3. The number of *derived variables* must be kept to a minimum
4. Due to potential side-effects, whenever possible avoid subroutines within.

The B-segments coincide with the notion of "critical code" established since the early days of parallel processing. The concept of derived variable is introduced here to mean any variable that is modified in the segment. In particular, any variable involved in a READ is a derived variable. *All derived variables must be broadcast.*

```
read(77,*) it,(igf(i,ib),i=1,nblock)
do  40 , j=1,kix
     p(j,ib) = a(j,ib)**2.0 * rho(j,ib) / gammar(nsp)
     e(j,ib) = P(j,ib)/ (gammar(nsp -1.0) +0.5*rho(j,ib)*u(j,ib)
40 continue
```

B1

Derived variables: it, igf, p, e

```
read(77,*) it,(igf(i,ib),i=1,nblock)
call energy(e, p, a, rho, gammar)
```

B2

Derived variables: it, igf, e (?) , p (?), a(?), rho(?), gammar(?)

FIG. 5.1. *In B1 it is possible by the syntax analyzer to detect the derived variables while in the second case, B2, it all depends on the called subroutines*

Figure 5.1 shows two B-segments showing the derived variables. Detecting derived variables can very tricky. Consider for example the case when there are routines within the B-segment. In some cases, it is very difficult to determine the *output* of a routine. Some routines in PAB3D prototype have more than 200 arguments. Proper documentation (rarely available in practice), might provide useful information. This can be complemented with the use of good compilers that provide cross-references which facilitates the detection of derived variables.

If the subroutine within contains side-effects, i.e. in the case it modifies non-local variables, the situation becomes extremely complicated. A possibility is to make the routine in-line, and let the compiler do the work, but even in this situation other difficulties may arise.

Fortunately, PAB3D it is well designed in this respect: variables in commons are explicitly passed as arguments, avoiding altogether the possibility of side-effects. However, this lead to another problem: the existence of dummy arrays.

Often the dimension of an array is provided only for checking the syntax of code. The real dimension, might be adjustable (passed as argument), or determined at some higher level common. Thus if a local variable is declared as *real a(1)*, the declaration might not represent its actual dimension, and thus one has to be careful at broadcast time.

Following a B-segment, as an source include file, the M1 source provides a '*.bct' file which contains the actual broadcast MPI commands. In many cases, these are generated automatically from syntax analyzers outputs with a fixed format as shown in Figure 5.2. Segment characteristics for a higher lever routine is show in Figure 5.3. Message lengths for arrays are determined from static data and thus the

```
 XX =  nblk*nsec*6*20*nprt*nzon
 call MPI_BCAST(ibcf,XX,MPI_INTEGER, MASTER,MPI_COMM_WORLD,
+       ierr)

 XX = nblk*nzon*ngt
 call MPI_BCAST(ibf,XX,MPI_INTEGER, MASTER,MPI_COMM_WORLD,
+       ierr)

 XX = nblk* ( 21+2*npcmx+1)
 call MPI_BCAST(iptf,XX,MPI_INTEGER, MASTER,MPI_COMM_WORLD,
+       ierr)

 XX = jkmx*(ncsp)+1
 call MPI_BCAST(q0s,XX,MPI_REAL, MASTER,MPI_COMM_WORLD,
+       ierr)
```

FIG. 5.2. *Automatically generated '*.bct' file*

Segmt.	Length	Der. Var	Subrtn.	I/O units	BCT file
B1	192	49		5	So13-M2B1
G1	244				
B2	32	30		98,99	So13-M1B2
G2	10		rinput		
B3	45	36		7	So13-M1B3
B4	24	19		97	So13-M1B4
G3	145		zonm inidct init jkbar **solver**		
B5	3		outf1		

FIG. 5.3. *Segment characteristics for high level routine*

ordering of the broadcasts calls is not important. In Section 6, we are going to retake this topic again.

The executable staments that form the complement of the B-segments, are the G-segments. These run in all process. Figure 5.4 shows the actual division of the most significant parts of the code in phase M1.

5.2. Phase M2: Communication. It is necessry to understand a little more about the global structure of the code, in order to describe the communication and computation subsystems. The phase M1, described above has carried the parallelization up to *solver*. Solver does a few *global* iterations. This a convenience in PAB3D to define breaking points in the compututation, after which partial solutions are output, and partial residuals are computed.

Figure 5.7 illustrates the buffer for exchanging data, *qbuf*, and processors P1 and P6 involved in the

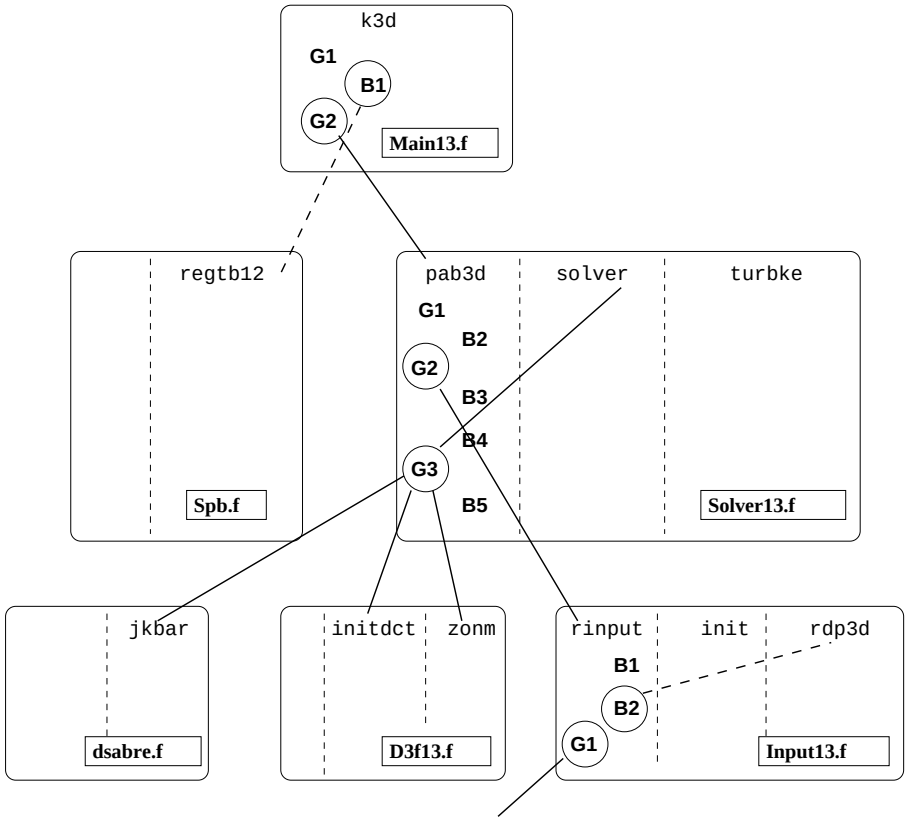

FIG. 5.4. *Segmentation for M1. Single processor calls are shown as dashed lines*

activity. The buffer consists of the cells of the pieces, create off-line using the patcher. Each block (process) is responsible for sending and receiving the pieces which connects it with another block or boundary. Referring to the figure, P1 sends pieces 10 and 25 while P6 receives them.

For each global iteration, *solver4* is called once; it it here where the core computations take place. The routine *solver4* consists of nested loops as follows: time loop, zone loop, and (a sequence of) block loops. It is at the lower level (block loop) where the process must work independently. However, before this can happen we must make sure tbat they have received all the pertinent information from other process. And this is done through the COMMSYS.

COMMSYS consist of two parts which interact exclusively with the arrays of the patching system. It is designed as a system of include files which contain its own databases. The include files named with lower cases contain only declarations/definition while the upper case ones contain executable statements. They are:

- **commsyspar.h** contains three parameters, for the commsys.h arrays: maximum number of blocks, maximun pieces per block, and maximun total pieces. While this information can be obtained directly from the PAB3D commons, leaving these as independent parameters provide more flexibility. The dependencies might be stated in the makefile.
- **commsys.h** declares the arrays. The global arrays correspond to the global piece numbers:

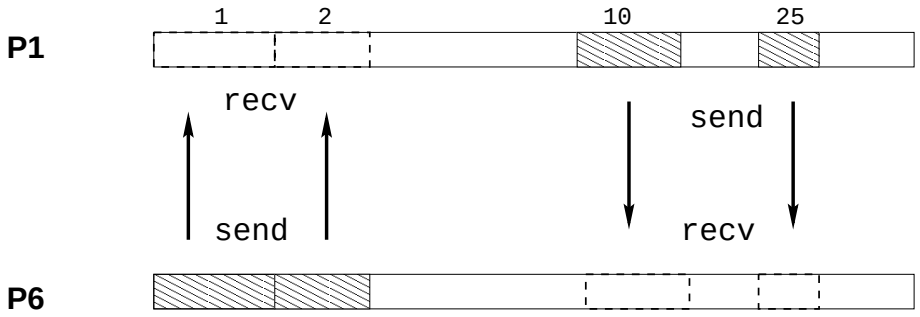

FIG. 5.5. *Two copies of* **qbuf** *in processes P1 and P6*

add_piece: (scalar) piece addres in qbuf.
from_piece: sending block of piece
recv_piece: receiving block of piece,

and the local arrays to the local block information:
n_piece: number of pieces of the block
send_piece: global piece that the block send
recv_piece: global piece the the block receives.

- COMM_GLOBAL.h : the global arrays are set according to runtime information. No actual MPI calls are issued here.
- COMM_LOCAL.h: Non-blocking receives and blocking sends are issued here.

In addition, the file commsys.h contain arrays declarations pertaining to communications request and status. Since the number of communication sending receiving messages depends on the number of pieces per block (a parameter in commsyspar.h), it was natural to include this type of declarations here.

Figure 5.6 shows the matching of the executables COMM_GLOBAL and COMM_LOCAL to the Version 13 patching subsystem.

5.3. Phase M3: Computation. As we mention the innermost loop in solver4 consists of block loops. In each of the cases, the index variable of the loop is called 'ib'; for simpicity we will to these as ib-loops.

Early in solver4, the ib-loop that invokes the communnication system has been executed as differenc instances. Subsequent ib-loop do the computations on the blocks (which we can assure that they contain the proper informacion since it has been exchanged through qbuf). Similarly, we execute separate

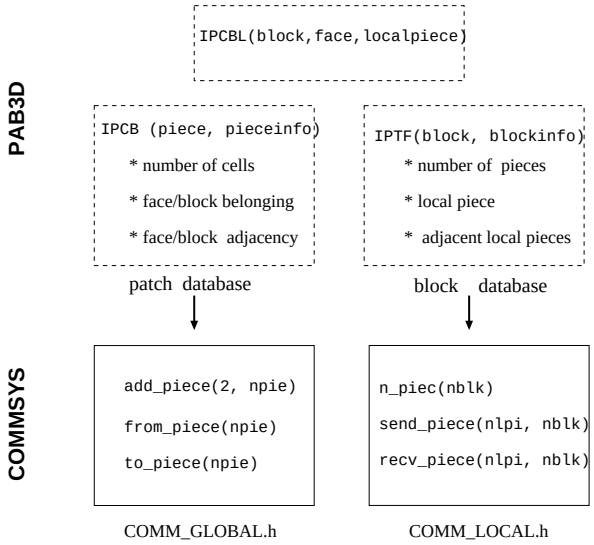

FIG. 5.6. *Relation between the databases of PAB3D and those of COMMSYS*

instances of this. The computational routines, which work with the patching information, will find the required data in qbuf.

Using the newly provided information, the instance i of loop ib, will run in processor i-1. All communication has been taken care in step M2, so there no communication commands added to the system at this stage. The only interphase with MPI is for self-identification, but this is already done in an earlier phase. Self-identification is simply using a parameter.

5.4. Phase M4: Updating. The final phase is the updating of the solution, at the end of the global iterations for output. This the reverse of an broadcast operation, a gather operation, in which block data is passed to the processor in charge of the output.

6. Status, Further Work and Conclusions. As we have been able to compute correct residuals in 9 different processors and bring them to a designated root processor, we are very optimistic about the successful completion of the project.

There are, however, a few things that need to be perfected. Here is a partial list which involve a different degrees of effort. The experiences learned here should provide clues on the design of future versions of PAB3D.

6.1. Timely Messaging. In an environment where computational loads from blocks vary widely, as in the case of our prototype problem as discussed in Section 2, it is fundamental that messages arrive in the order in which they are posted.

LAM/MPI and MPI provide a number of options, such as tagging and specific implementation of point-to-point communication commands. In order to use the optimal method for our purposes, a test-problem

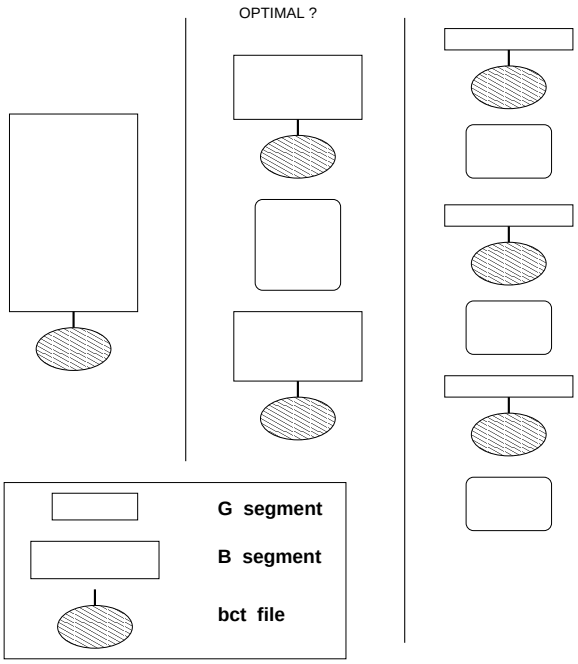

FIG. 6.1. *Optimal segmentation*

must be properly designed. A prototype, such as the one we worked on, answered the question whether the code, with realistic data, *can* run in parallel or not. A test-problem is needed to guide further implementation decisions.

6.2. Broadcast optimization. The conditions of on the construction of a segment partition described in Section 5 may seem a little restrictive. However, they define a whole hierarchy of partitions: from the trivial partitions like making the whole code either G or B or its opposite, that is making each I/O operation B and the complement G. Clearly, this properly defines an optimization problem, with a parameter being the size of the broadcast (number of process). This idea is illustrated in Figure 6.2.

Certainly, the above can be a quite elaborated problem. However, the code in its present state admits, in principle, a straightforward improvement, which only involve the broadcast files.

Due to provisions for future robustness, as we explain in Section 5, the message lengths of most derived broadcast variables, especially dummy arrays, involve the *declared* dimension. These lengths can be changed to *actual* lengths.

The organization of the *bct* files, Figure 5.1 has been purposedly left in a uniform format to allow easily a transition for this provision. Finding the correct value for the size of the message might involve significant work, but should be relatively easy for someone who knows the code well. Nonetheless, a word of caution is necessary: some of the actual lenghts are broadcast as derived variables themselves. Thus, it might be required to rearrange the *bct* so as to let them be broadcast before they are used.

6.3. Memory Distribution. In Section we explain the reasons why we decided to start with the full block approach. For efficiency, it is imperative that total memory requirements must be reduced by using the shrunken block model.

This is a major change of the code, as it modifies the core data structures. This improvement should leave the way open for distributed I/O.

7. Acknowledgements. The first author would like to thank David Keyes of ICASE and Old Dominion University for his support and for making the project possible.

REFERENCES

[1] K. Abdol-Hamid, *A multiblock/multizone code (PAB3D-v2) for the three-dimensional Navier-Stokes equations: Preliminary applications*, Tech. Report CR-182032, NASA, October 1990.

[2] ——, *Implementation of algebraic stress model in a general 3-d navier-stokes method (pab3d)*, Tech. Report CR-4702, NASA, 1995.

[3] K. Abdol-Hamid, J. Carlson, and B. Lakshmannan, *Application of Navier-Stokes code PAB3D to attached and separated flows for use with k-e turbulence model*, Tech. Report TP-3489, NASA, 1994.

[4] K. Abdol-Hamid, J. Carlson, and S. Pao, *Calculation of turbulent flows using mesh sequencing and conservatie patch algorithm*, Tech. Report 95-2336, AIAA, 1995.

[5] A. Ecer, J. Periaux, N. Satofuka, and S. Taylor, eds., *Parallel Comptuational Fluid Dynamics*, North Holland, January 1996. Proceedings of Parallel CFD conference, June 26-28, 1995, Pasadena, California.

[6] C. Fischberg, C. Rhie, R. Zacharias, P. Bradley, and T. DesSureauault, *Using hundreds of workstations for production running of parallel cfd applications*, in Parallel Computational Fluid Dynamics, A. Ecer, J. Periaux, N. Satofuka, and S. Taylor, eds., Stony Brook, North Holland, 1996, pp. 9–22.

[7] GDB/RBD, *MPI primer/ developing with LAM*, tech. report, Ohio Supercomputer Center, 1224 Kinnear Road, Columbus, OH 43212, November 1996.

[8] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI*, The MIT Press, 1994.

[9] Oak Ridge National Lab, *MPI: A message passing interface standard*, tech. report, University of Tenessee, 1995.